

El lenguaje de programación Java

Jordi Bríñquez Jiménez

PID_00161680

Índice

| | |
|---|----------|
| Introducción | 5 |
| Objetivos | 6 |
| 1. Java como lenguaje de programación estructurada | 7 |
| 1.1. Introducción a Java | 7 |
| 1.1.1. Características del lenguaje..... | 8 |
| 1.1.2. Palabras reservadas | 8 |
| 1.1.3. Convenciones en el nombre de variables y funciones | 9 |
| 1.1.4. El primer programa en Java | 10 |
| 1.2. Tipos de datos | 10 |
| 1.2.1. Números | 11 |
| 1.2.2. Caracteres | 12 |
| 1.2.3. Tipos enumerados | 13 |
| 1.2.4. Constantes | 13 |
| 1.2.5. El tipo booleano | 14 |
| 1.3. Operadores | 14 |
| 1.3.1. Operador de asignación | 15 |
| 1.3.2. Operadores aritméticos | 15 |
| 1.3.3. Operadores relacionales | 16 |
| 1.3.4. Operadores lógicos | 16 |
| 1.3.5. Operadores a nivel de bit | 17 |
| 1.3.6. Operadores equivalentes | 18 |
| 1.3.7. Operador condicional | 19 |
| 1.3.8. Precedencia de operadores | 20 |
| 1.4. Matriz y vectores | 22 |
| 1.5. Bloques de instrucciones | 23 |
| 1.5.1. Bloques condicionales | 23 |
| 1.5.2. Bloques iterativos | 25 |
| 1.5.3. Sentencia <code>break</code> | 28 |
| 1.5.4. Sentencia <code>continue</code> | 28 |
| 1.6. Funciones..... | 28 |
| 1.6.1. Definición de funciones | 29 |
| 1.6.2. Los parámetros de entrada | 29 |
| 1.6.3. El valor de retorno | 30 |
| 1.6.4. Un ejemplo de función | 30 |
| 1.6.5. La invocación de funciones | 31 |
| 1.7. Visibilidad de variables | 32 |
| 1.7.1. Variables locales | 32 |
| 1.7.2. Variables globales | 33 |

| | |
|---|-----------|
| 2. Java como lenguaje de programación orientado a objetos | 34 |
| 2.1. Definición de clases | 34 |
| 2.1.1. La clase | 34 |
| 2.1.2. Los atributos de una clase | 35 |
| 2.1.3. Métodos de una clase | 36 |
| 2.1.4. El método constructor | 36 |
| 2.1.5. El método destructor | 37 |
| 2.1.6. El resto de métodos | 37 |
| 2.1.7. Uso de la palabra reservada <code>this</code> | 38 |
| 2.1.8. Métodos estadísticos | 39 |
| 2.1.9. Sobrecarga de métodos | 40 |
| 2.2. Las relaciones entre clases | 41 |
| 2.2.1. Cardinalidad | 41 |
| 2.2.2. Nombre exacto | 41 |
| 2.2.3. Rango de valores | 42 |
| 2.2.4. Valores indefinidos | 42 |
| 2.2.5. Navegabilidad | 43 |
| 2.2.6. Roles | 43 |
| 2.3. Biblioteca de clases | 43 |
| 2.3.1. La clase <code>String</code> | 44 |
| 2.3.2. La clase <code>ArrayList</code> | 46 |
| 2.4. Las excepciones | 48 |
| 2.4.1. Creación de una excepción | 48 |
| 2.4.2. Lanzamiento de excepciones | 49 |
| 2.4.3. Tratamiento de excepciones | 50 |
| Resumen | 52 |
| Actividades | 53 |
| Solucionario | 54 |
| Glosario | 54 |
| Bibliografía | 55 |

Introducción

Este módulo pretende repasar los conceptos básicos de codificación adquiridos en otras asignaturas. En concreto, haremos un repaso de conceptos de programación descendente y prepararemos al estudiante de cara a los nuevos conceptos que se expondrán en esta asignatura.

Este módulo está organizado en dos grandes bloques:

- Una introducción al lenguaje Java como lenguaje de programación estructurada.
- La utilización del lenguaje Java como lenguaje orientado a objetos.

En el primer apartado, “Java como lenguaje de programación estructurada”, se pretende que los estudiantes reviséis la sintaxis básica de Java (tipos de datos, operadores, bloques condicionales e iterativos, funciones, etc.), y otros conceptos básicos como la visibilidad de las variables.

En el segundo apartado se introducen todos aquellos conceptos específicos de Java que nos permitirán trabajar con el paradigma de la programación orientada a objetos. Aparte, explicaremos conceptos de interacción con el usuario a partir de la pantalla y el teclado con el fin de poderlos aplicar a los nuevos conceptos que se expondrán en esta asignatura, con lo cual podremos pasar a crear nuestros primeros “programas orientados a objetos”.

Posteriormente completaremos el aprendizaje con la gestión de errores utilizando las excepciones o el tratamiento de ficheros.

Objetivos

Los objetivos de este módulo son principalmente dotar al estudiante de unos conocimientos básicos a fin de que pueda realizar las prácticas de la asignatura con cierta agilidad.

Se supondrán unos conocimientos mínimos de programación, ya que no es el objetivo de este módulo enseñar a programar, sino introducir los conceptos de la programación orientada a objetos a partir de un lenguaje de programación concreto (en este caso Java).

Asimismo, tampoco tiene la intención de ser un manual del lenguaje de programación Java; para ello existen muchos manuales (unos en papel y otros disponibles en la web) que cubren casi todas las necesidades de un programador.

1. Java como lenguaje de programación estructurada

1.1. Introducción a Java

El lenguaje de programación Java apareció a principios de los años noventa del siglo XX a raíz de un proyecto interno de la empresa Sun Microsystems que tenía como objetivo permitir al programador trabajar de manera más ágil, sin algunos de los problemas que presentaban algunos lenguajes de programación como C/C++. Este lenguaje prometía en sus inicios la posibilidad de escribir el código una única vez y poder ejecutarlo en muchas plataformas diferentes. Inicialmente se utilizaba como lenguaje para escribir pequeños programas que se incrustaban en las páginas web (los *applets*) y poco a poco ha ido ganando terreno hasta la programación de grandes aplicaciones empresariales, así como de páginas web enteras y dispositivos móviles.

El lenguaje Java se creó teniendo en cuenta cinco principios:

1. Utilizar la metodología de la orientación a objetos
2. Permitir la ejecución del mismo programa directamente en diferentes sistemas operativos
3. Permitir la utilización de la red de manera sencilla
4. Permitir la ejecución de código remoto de manera segura
5. Ser sencillo de programar.

Para alcanzar el segundo objetivo, los ingenieros de Sun crearon un compilador que transforma el código fuente en un lenguaje que se denomina *bytecode*. Éste es un lenguaje intermedio entre el código fuente y el lenguaje máquina, que posteriormente una máquina virtual ejecuta en la plataforma correspondiente.

Esta máquina virtual permite una independencia de la plataforma sobre la que se ejecuta el código; por lo tanto, el programador de Java queda liberado de la tarea de hacer compatible el código en todas las plataformas en las que quiere que éste se ejecute, ya que es Sun Microsystems la responsable de este trabajo. Es el concepto de “compilar una vez y ejecutarlo en todas partes”^{*} que Sun utilizó en el lanzamiento del lenguaje Java.

Desde 1996, cuando salió la versión 1.0, hasta la actualidad (ya existe la versión 6.0), han aparecido varias versiones, y varias subversiones para cada una de ellas que arreglaban posibles errores. Pero no sólo han surgido versiones para el desarrollo de aplicaciones de escritorio, sino que también existe desde hace tiempo una versión reducida y adaptada, tanto del lenguaje como de la máquina virtual, para ejecutar aplicaciones en dispositi-

Máquina virtual

Una máquina virtual es un programa que es capaz de ejecutar otros programas y de garantizar su seguridad.

^{*} En inglés, *compile once, run everywhere*.

vos móviles (como teléfonos y agendas personales), fotocopadoras u otros aparatos electrónicos.

1.1.1. Características del lenguaje

El lenguaje de programación Java puede estar compilado casi en cualquier sistema.

El lenguaje Java se considera un lenguaje de alto nivel. Java facilita mucho las tareas de programación, ya que permite programar sin tener en cuenta la plataforma en la que ejecutan las aplicaciones.

Recordad

Este nivel indica el grado de proximidad del lenguaje con la máquina.

Algunas de las características que podemos mencionar del lenguaje Java son las siguientes:

- Un núcleo del lenguaje simple.
- Está orientado a la programación utilizando el paradigma de la orientación a objetos.
- Dada la biblioteca de objetos que se proporciona con el compilador, se transforma en un lenguaje muy potente.
- Acceso y gestión de la memoria sencillo gracias al reciclaje de memoria*.
- Variables locales, globales y de clase.
- Seguridad de tipos.
- Sobrecarga de operadores y métodos.

* En inglés, *garbage collection*.

Esta potencia que ofrece, junto con la facilidad de aprendizaje, han provocado que Java sea uno de los lenguajes de programación más utilizados hoy en día, tanto en entornos académicos como en entornos empresariales.

Aunque el lenguaje de programación Java es un lenguaje pensado para el paradigma de la programación orientada a objetos, no lo podemos clasificar como un lenguaje orientado a objetos “puros”, dado que sus tipos básicos no son objetos (aunque en las nuevas versiones los compiladores los transforman en objetos directamente, sin la necesidad de realizar conversiones explícitas).

! Podéis ver los lenguajes orientados a objetos puros e híbridos en el módulo “Clases y objetos” de esta asignatura.

1.1.2. Palabras reservadas

En un lenguaje de programación, una **palabra reservada** es aquella que no se puede utilizar como identificador porque ya se utiliza dentro de la gramática del propio lenguaje.

Esto significa que no podemos utilizar ciertas palabras para dar nombres ni a variables ni a funciones. Veamos cuáles son las palabras reservadas del lenguaje Java.

| | | | | |
|----------|----------|------------|-----------|--------------|
| abstract | continue | for | new | switch |
| assert | default | if | package | synchronized |
| boolean | do | goto | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

Por lo tanto, no podremos utilizar ninguno de estos nombres para identificar variables o funciones dentro de nuestro código. Veremos el significado de la mayoría de estas palabras a lo largo de este apartado.

1.1.3. Convenciones en el nombre de variables y funciones

En Java, como en cualquier lenguaje de programación, hay que seguir unas normas para nombrar las variables y las funciones. En caso de que no se sigan, se producirá un error de compilación.

Estas reglas son las siguientes:

- El primer carácter debe ser una letra o el carácter ‘_’.
- Los caracteres que sigan a continuación pueden ser tanto letras como números y caracteres ‘_’.
- Las mayúsculas y las minúsculas se consideran símbolos diferentes.
- Se pueden utilizar caracteres como letras con tilde, la ‘ç’ o la ‘ñ’, porque Java utiliza la tipografía Unicode, pero es altamente desaconsejado.
- No se pueden utilizar espacios en blanco.
- El nombre no puede ser una palabra reservada.

Enlaces de interés

Para conocer todos los caracteres Unicode y sus implicaciones podéis visitar la página:
<http://www.unicode.org>

Estas reglas son las que impone el lenguaje, pero para hacer lo más inteligible posible la utilización de las variables y funciones, es recomendable seguir los criterios siguientes:

- Escribir los nombres en minúsculas.
- Utilizar nombres descriptivos.
- Evitar nombres que se puedan confundir con facilidad.
- Escribir la primera letra de las palabras “intermedias” en mayúsculas.

Veamos unos ejemplos de identificadores correctos, incorrectos y no recomendados:

| Nombre o identificador | ¿Es correcto? | Comentario |
|------------------------|-----------------|---|
| ventas | Sí | |
| aux2 | No recomendable | Es mejor utilizar nombres más descriptivos. |
| 1Articulo | No | No puede empezar por un número |
| const | No | Es una palabra reservada en Java |
| ventasEnero | Sí | |
| ventasenero | Sí | |

Cabe tener en cuenta que el lenguaje Java diferencia las mayúsculas de las minúsculas*. Por lo tanto, los dos últimos identificadores son diferentes y no identifican el mismo elemento. Hay que ir con cuidado y utilizar unos criterios regulares durante todo el programa para evitar confusiones.

* Un lenguaje que diferencia mayúsculas y minúsculas se dice que es *case sensitive*.

1.1.4. El primer programa en Java

Llegado a este punto y antes de continuar con las peculiaridades de Java, quizá es conveniente ver qué aspecto tiene un programa sencillo para ir acostumbrándonos a este lenguaje.

```
public class HelloUOC {
    public static void main(String[] args) {
        System.out.println("Welcome to UOC")
    }
}
```

Observación

Aunque hay palabras que no entendemos (como `public` o `static`), sí que se puede entender aproximadamente el funcionamiento del programa.

Este programa únicamente escribe un mensaje de bienvenida por pantalla. Como podéis ver, hay elementos que pueden resultar extraños, pero iremos explicando su significado poco a poco a lo largo de este apartado.

1.2. Tipos de datos

En nuestros programas en Java, al igual que en cualquier otro lenguaje de programación, deberemos almacenar nuestros datos en variables. Los tipos de datos que éstas pueden representar tendrán un formato lógico, numérico y carácter (también denominado *alfanumérico*). Dentro del formato numérico podemos representar números enteros y números reales. Veamos sus características y cómo se pueden representar en lenguaje Java.

1.2.1. Números

Como ya hemos comentado, los tipos numéricos se pueden dividir en dos grandes grupos: números enteros y números en coma flotante. Veamos

qué tipos de datos tenemos y el rango de los valores que éstos pueden almacenar.

Números enteros

Los números enteros son los que utilizamos normalmente para contar objetos, por ejemplo: tres manzanas.

En Java tenemos diferentes tipos de datos para representar los números enteros. La principal diferencia es el rango numérico que podemos representar con los diferentes tipos. En la tabla siguiente los vemos todos:

| Tipos | Medida en bytes | Rango | Ejemplos |
|-------|-----------------|----------------------------|-----------------|
| Byte | 1 | $-2^7 \dots 2^7 - 1$ | -100000, 345678 |
| Short | 2 | $-2^{15} \dots 2^{15} - 1$ | 32769, 0xffea |
| Int | 4 | $-2^{31} \dots 2^{31} - 1$ | 234, -1500 |
| Long | 8 | $-2^{63} \dots 2^{63} - 1$ | -64323, 0xaffaf |

Podemos representar los números enteros utilizando diferentes notaciones:

- Notación **decimal**, base 10: 1234, 925, 12345678901234
- Notación **octal**, base 8 (empiezan con 0): 02, 0123, 0534
- Notación **hexadecimal**, base 16 (empiezan con 0x): 0xa3, 0xffffa

El modificador `unsigned` denota que los valores que se almacenan son únicamente positivos. Por lo tanto, tenemos el doble de espacio y podemos aprovechar el espacio reservado a los números negativos para almacenar números mayores.

De todos los tipos de los que disponemos para representar los enteros, el tipo `int` será el que utilizaremos normalmente, pero hay que tener en mente que los otros existen por si debemos construir algún programa que los requiera.

Más adelante veremos qué operadores podemos utilizar para trabajar con los números enteros, pero para que os hagáis una idea, serán todas las operaciones matemáticas más comunes (suma, resta, multiplicación, división, etc.).

Para números mayores

Si necesitáramos números mayores, deberíamos utilizar clases como, por ejemplo, `BigInteger`, que permiten almacenar números de cualquier tamaño.

Podéis ver los operadores para trabajar con los números enteros en el subapartado 1.3 de este módulo.

Números en coma flotante

Estos números, normalmente denominados decimales, son aquellos que se utilizan para denotar cosas que no pueden ser medidas únicamente utilizando unidades, por ejemplo: 1,5kg de manzanas, 3,25 €.

Igual que nos sucede con los números enteros, en Java tenemos más de una manera de representar los números reales. En la siguiente tabla vemos un resumen de los tipos básicos disponibles.

| Tipos | Tamaño en bytes | Rango |
|---------------------|-----------------|---|
| <code>float</code> | 4 | $3.4\text{E}-38 \dots 3.4\text{E}+38$ y $-3.4\text{E}-38 \dots -3.4\text{E}+38$ |
| <code>double</code> | 8 | $1.7\text{E}-308 \dots 3.4\text{E}+308$ y $-1.7\text{E}-308 \dots -1.7\text{E}+308$ |

Como podéis ver en esta tabla, se pueden representar números muy grandes y, al mismo tiempo, números muy pequeños con el mismo tipo de dato. En este caso las posibles notaciones son dos:

- Notación decimal: 0.1 12.25 -0.002341
- Notación científica: 2.3E+5 0.123E-32 4.98E+5.4

Del mismo modo que con los números enteros, más adelante también veremos con más detalle las operaciones que podemos realizar con los números reales.

Notación de Java

La notación que utiliza Java para la representación de números decimales es la propuesta por el IEEE 754.

Notación científica

La notación científica sirve para abreviar en la escritura. Por ejemplo:

$$2.3\text{E}+5 = 2.3 \cdot 10^5$$

$$0.123\text{E}-32 = 0.123 \cdot 10^{-32}$$

1.2.2. Caracteres

El **carácter** es un tipo de datos especial en Java que, a pesar de representar una letra o símbolo del alfabeto (recordemos que antes hemos comentado que Java utiliza Unicode), suele ser tratado internamente como un número. El nombre del tipo de datos es `char`.

Esta característica proviene del hecho de que los alfabetos en un ordenador no son más que codificaciones que utilizan 2 bytes (en vez del byte que utiliza la codificación ASCII) para almacenar las letras. El modo para que el compilador pueda interpretar estos caracteres como tales, y no como instrucciones de nuestro programa, es ponerlos entre comillas simples.

Los valores válidos para un elemento de tipo `char` suelen ser letras (mayúsculas y minúsculas), números y signos de uso común como, por ejemplo, operadores matemáticos, símbolos de puntuación y otros caracteres especiales como el tabulador (`'\t'`), para los cuales habrá que hacer uso de combinaciones de caracteres porque no tienen una representación gráfica en la codificación ASCII. Como Java utiliza la codificación Unicode, se pueden representar cualquiera de los caracteres existentes.

Las principales operaciones que podemos llevar a cabo sobre un elemento de tipo `char` son las comparaciones, pero dado que internamente se almacena como un entero de 2 bytes, también se pueden hacer operaciones aritméticas. Aun así, no es aconsejable, por no confundir conceptos, y además, porque muchas veces proporciona resultados inesperados.

El uso de Unicode

La utilización de la codificación Unicode se debe al hecho de que no todos los idiomas utilizan los mismos caracteres, y con los 256 caracteres que contiene la codificación ASCII no había suficiente para representar cualquier alfabeto (cirílico, chino, etc.).

¿Sabíais que...?

El tipo carácter se puede utilizar como número, pero no se aconseja debido a su poca capacidad de almacenamiento.

1.2.3. Tipos enumerados

Los **tipos enumerados** son un tipo especial de datos que permiten crear **tipos de datos propios**, siempre que sepamos qué valores queremos que este nuevo tipo acepte en tiempo de compilación.

Los tipos enumerados se introdujeron en la versión 1.5 del lenguaje Java.

Este tipo de datos suelen utilizarse para asociar valores constantes a nombres y hacer más inteligible el código. Un ejemplo claro de tipo enumerado serían los días de la semana o los meses del año.

```
enum days {SUN, MON, TUE, WED, THU, FRI, SAT};

enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
             OCT, NOV, DEC};
```

Las enumeraciones en Java se tratan como las cadenas de texto que se utilizan para identificar los elementos. Se puede obtener su traducción a un valor de tipo entero que nos indique la posición, pero para nosotros eso resulta completamente transparente. Por lo tanto, las podemos utilizar como si fueran nuevos tipos básicos.

```
days d = SUN;
...

if (d == THU) {
    ...
}
```

Observación

La notación que utilizaremos aquí no es exactamente la propuesta porque la mayoría de las enumeraciones se definirán dentro de una clase. Retomaremos esta cuestión cuando veamos las clases.

1.2.4. Constantes

Las constantes por sí mismas no son ningún tipo de datos, pero vale la pena mencionarlas en este apartado.

Una **constante** es únicamente un valor que podemos asignar o comparar a un tipo de datos determinado, pero que nunca podrá variar durante la ejecución del programa.

Para la declaración de las constantes hay que utilizar la palabra `final` con el fin de denotar que una vez asignado el valor ya no se podrá cambiar; y posteriormente el tipo de dato que almacenaremos.

Si se conoce el valor de la constante antes de la compilación, se suele asignar en el momento de la declaración. Si, por el contrario, el valor depende de al-

gún parámetro (recordemos que una vez asignado no se puede cambiar) se puede asignar en cualquier momento.

```
final int DAYS_OF_THE_WEEK= 7;  
final char ENDLINE      =  '\n';
```

Los nombres de las constantes se suelen escribir en letras mayúsculas

1.2.5. El tipo booleano

El tipo de dato **booleano** es aquel que puede tomar los valores 'cierto' o 'falso'.

Este tipo de datos se utiliza en operaciones booleanas, como las operaciones AND y OR lógica, condiciones de finalización de iteraciones, etc.

Al contrario que en otros lenguajes, en Java hay un tipo de datos que nos ayuda a representar este concepto y no es necesario utilizar enteros para su representación. Los posibles valores que puede tomar son `true` ('cierto') y `false` ('falso').

1.3. Operadores

Los **operadores** son símbolos del lenguaje que nos permiten realizar operaciones con los elementos implicados (también denominados **operandos**). Los operadores pueden ser de distinto tipo según su funcionalidad y el número de operandos que reciben.

Un concepto que hay que tener claro antes de empezar a ver qué operadores tiene el lenguaje Java es el concepto de *expresión*. Una **expresión** es una combinación de valores, funciones, operadores y otras expresiones que, utilizando las reglas de precedencia de los operadores que las forman, generan un resultado (otra expresión).

Otro tema que aprenderemos en este subapartado es la precedencia que tienen los operadores. A medida que las expresiones sean más complejas, habrá que saber cuál es la precedencia que tiene cada operador con el fin de conseguir los resultados esperados.

Podemos dividir los operadores aritméticos en 2 grupos según el número de operandos que utilizan. Así, tenemos:

- para los operadores que utilizan un operando:

```
operator op1
```

- y para los operadores que utilizan dos operandos (el resto):

```
op1 operator op2
```

1.3.1. Operador de asignación

Los **operadores de asignación** se utilizan para asignar valores a las variables.

En Java, este operador se simboliza con un símbolo de igual ('='). Las asignaciones pueden ser entre una variable y un valor o entre dos variables:

```
int var1;  
int var2;  
var1 = 3;  
var2 = var1;
```

En las asignaciones entre dos variables hay que evitar que se pierda información en la asignación o que puedan producirse desbordamientos:

```
short s;  
int i = 450000;  
s = i; // s cannot handle such big number
```

Efectos no deseados

Se pueden dar efectos no deseados si intentamos asignar un valor mayor al valor máximo representable en un tipo de datos.

1.3.2. Operadores aritméticos

Los **operadores aritméticos** son aquellos que nos permiten realizar operaciones aritméticas con números, bien sean enteros o reales.

En la siguiente tabla tenéis una descripción de los operadores que se pueden utilizar y los tipos de datos sobre los que se pueden utilizar:

| Operador | Significado | Tipo de dato | Ejemplo |
|----------|-----------------|--------------|--------------|
| - | Cambio de signo | Numérico | -3, -4.2 |
| + | Suma | Numérico | 3+8, 3.4+1.5 |
| - | Resta | Numérico | 9-4, 2.9-5.3 |
| * | Multiplicación | Numérico | 3*4, 3.0*4.1 |
| / | División | Numérico | 3/4, 4.2/9.3 |
| % | Módulo | Entero | 250%23 |

También se pueden realizar operaciones entre diferentes tipos de datos (por ejemplo, podríamos sumar uno `float` con uno `int`). Ahora bien, siempre hay que tener en cuenta que la división entre dos números enteros puede producir un número no entero, de manera que habría que almacenar el resultado en una variable de tipo `float` con el fin de no perder información.

Hay que puntualizar que cualquier operación que resulte en un valor entero (suma, resta, multiplicación y módulo de números enteros) siempre será de tipo `int` si el valor resultante puede ser representado por entero; en caso contrario, se utilizará el tipo `long`.

1.3.3. Operadores relacionales

Los **operadores relacionales**, también denominados **comparativos**, son símbolos que nos ayudan a expresar relaciones entre dos entidades, como por ejemplo “mayor que”, “menor o igual que”, etc.

El resultado de las comparaciones será un valor booleano, cierto o falso, según sea el resultado.

Las operaciones relacionales que Java nos ofrece son:

| Operador | Significado | Ejemplo (cierto) | Ejemplo (falso) |
|----------|-------------------|------------------|-----------------|
| > | Mayor que | 5 > 3 | -99 > 13 |
| >= | Mayor o igual que | 5 >= 3.2 | 6 >= 19 |
| < | Menor que | -3 < 2 | 4 < 1 |
| <= | Menor o igual que | -3.9 <= 9.8 | 9.8 <= -5 |
| == | Igual que | 3 == 3 | 3 == 4 |
| != | Diferente de | 2 != 7 | 2 != 2 |

Como podéis ver, se pueden comparar elementos de diferentes tipos numéricos. El compilador se encarga de realizar las conversiones correspondientes para que los operandos sean del mismo tipo.

1.3.4. Operadores lógicos

Los **operadores lógicos** son los que utilizamos para operar con valores booleanos. El resultado de estas operaciones es siempre un valor booleano.

En la tabla siguiente se presentan algunos ejemplos de operadores lógicos:

| Operador | Significado | Ejemplo |
|----------|-------------|---|
| && | AND lógica | <code>expresion1 && expresion2</code> |
| | OR lógica | <code>expresion1 expresion2</code> |
| ! | NOT lógica | <code>!expresion1</code> |

Una curiosidad sobre los operadores lógicos && y || es que evalúan las expresiones por orden (lo que no sucede con los otros operadores), de modo que el resultado de la operación, dejamos de valorar la expresión. Es decir, en el ejemplo anterior, el operador && evaluaría primero `expresion1` y, en caso de que fuera cierta, como todavía no podemos garantizar el resultado, evaluaría el segundo operando (`expresion2`). En caso contrario, se dejaría de evaluar la expresión y se daría directamente el resultado de 'falso'. En el caso de una OR, la expresión se deja de evaluar cuando uno de los operandos produce un resultado de 'cierto'.

La operación XOR lógica no está contemplada dentro del lenguaje Java, pero se puede escribir fácilmente como:

```
(op1 || op2) && !(op1 && op2)
```

1.3.5. Operadores a nivel de bit

En un ordenador todos los datos se transforman en ceros (0) y unos (1). Por lo tanto, se necesitan algunas operaciones para trabajar a nivel de bit con los valores de los datos.

De las 6 operaciones existentes para trabajar a nivel de bit, hay tres formatos diferentes según la operación.

Trabajar a nivel de bit...

... significa que las operaciones se ejecutan bit a bit y que el resultado de la operación es el valor de concatenar todos los bits.

| Operador | Significado | Ejemplo (unsigned short) |
|----------|----------------------|--|
| & | AND de bits | <code>5&3 = 1</code> , <code>64&63 = 0</code> |
| | OR de bits | <code>5 3 = 7</code> , <code>170 85 = 255</code> |
| ^ | XOR de bits | <code>5^3 = 6</code> , |
| ~ | complemento a 1 | <code>~5 = 65530</code> , |
| << | desp. a la izquierda | <code>5<<3 = 40</code> , <code>1<<8 = 512</code> |
| >> | desp. a la derecha | <code>5>>3 = 0</code> , <code>147>>5 = 4</code> |

Para entender un poco más el funcionamiento de estas operaciones, veremos cómo funcionan paso a paso.

Los operandos que utilizaremos para estos ejemplos serán números enteros de 8 bits sin signo con el fin de facilitar los cálculos. Éstos son:

- $175 = 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10101111$ (en base 2)
- $49 = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^0 = 00110001$ (en base 2)

Vamos a ver cómo funcionaría la operación $175 \ \& \ 49$:

$$\begin{array}{r}
 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 = 175 \\
 \& \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 = 49 \\
 \hline
 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 = 33
 \end{array}$$

Estos métodos utilizan las tablas de verdad como las operaciones lógicas, que podemos resumir en:

| | | |
|-----|---|---|
| AND | 0 | 1 |
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| | | |
|----|---|---|
| OR | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| | | |
|-----|---|---|
| XOR | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 1 | 0 |

| | |
|-----|---|
| NOT | |
| 0 | 1 |
| 1 | 0 |

Las operaciones de desplazamiento de bits son también muy sencillas: se efectúa un desplazamiento de todos los bits a la derecha o a la izquierda.

Veamos un ejemplo de esto para eliminar dudas. Haremos $175 \gg 3$, $49 \ll 1$, y $-8 \ggg 2$:

$$\begin{array}{r}
 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 = 175 \\
 \gg +3 \\
 \hline
 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 = -11
 \end{array}$$

$$\begin{array}{r}
 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 = 49 \\
 \ll 1 \\
 \hline
 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 = 98
 \end{array}$$

$$\begin{array}{r}
 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 = -8 \\
 \ggg 2 \\
 \hline
 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 = 62
 \end{array}$$

Desplazamiento del bit

Como podéis ver, la operación de desplazar a la derecha equivale a dividir por 2^n , y la de desplazar a la izquierda, a multiplicar por 2^n (donde n es el número de posiciones).

1.3.6. Operadores equivalentes

Los operadores que acabamos de ver sirven para evaluar una o dos expresiones (dependiendo del operador) y devuelven el valor resultante.

En Java existen una serie de operadores que nos permiten realizar algunas operaciones sin haber de escribir tanto, pero que a menudo ayudan a hacer el código más claro.

Los operadores ++ y --

Los operadores ++ y -- se denominan **postincremento** y **postdecremento** cuando se escriben a la derecha de la variable y suelen utilizarse muchas veces en los incrementos de los bloques iterativos.

Cuando se escriben a la izquierda de la variable, se denominan operadores de **preincremento** y **predecremento**.

| Operador | Ejemplo | Notación larga (equivalente) | Resultado |
|----------|-----------------|------------------------------|-----------|
| += | i = 3; i += 5 | i = 3; i = i + 5 | i = 8 |
| -= | i = 3; i -= 5 | i = 3; i = i - 5 | i = -2 |
| *= | i = 3; i *= 5 | i = 3; i = i * 5 | i = 15 |
| /= | i = 3; i /= 5 | i = 3; i = i / 5 | i = 0 |
| %= | i = 3; i %= 5 | i = 3; i = i % 5 | i = 3 |
| &= | i = 3; i &= 5 | i = 3; i = i & 5 | i = 1 |
| = | i = 3; i = 5 | i = 3; i = i 5 | i = 7 |
| ^= | i = 3; i ^= 5 | i = 3; i = i ^ 5 | i = 6 |
| <<= | i = 3; i <<= 5 | i = 3; i = i << 5 | i = 96 |
| >>= | i = 3; i >>= 5 | i = 3; i = i >> 5 | i = 0 |
| >>>= | i = 3; i >>>= 2 | i = 3; i = i >>> 2 | i = 0 |
| ++ | i = 3; i++ | i = 3; i = i + 1 | i = 4 |
| -- | i = 3; i-- | i = 3; i = i - 1 | i = 2 |

No todas estas abreviaciones suelen ser de uso común dentro de los programas, pero es interesante conocerlas para poder entender el código de otros programadores.

1.3.7. Operador condicional

En Java existe un operador especial, ya que recibe tres operandos, con el formato siguiente:

```
op1 ? op2 : op3
```

Este operador evalúa `op1` (que debe ser una expresión booleana) y dependiendo del resultado devuelve como valor resultante el valor de la expresión `op2` (en caso de que `op1` sea cierto) o el valor de la expresión `op3` (en caso de que sea 'falso').

Puede llegar a ser complejo entender el funcionamiento de este operador, pero quedará claro con la secuencia de instrucciones siguiente:

```
int i = 0;           // value of i -> 0
i = (i > 0) ? -3 : 3; // value of i -> 3
i = (i > 0) ? -3 : 3; // value of i -> -3
```

Como seguramente habréis deducido, el uso de este operador es equivalente al siguiente código:

```
i = 0;
if (i > 0) {
    i = -3;
}
else {
    i = 3;
}
if (i > 0) {
    i = -3;
}
else {
    i = 3;
}
```

En el subapartado 1.5.1 veremos con detalle cómo funciona el bloque condicional if-else.

1.3.8. Precedencia de operadores

Hasta ahora las expresiones que hemos escrito han sido muy sencillas y no hemos tenido ningún problema para averiguar su resultado final, pero se pueden escribir expresiones tan complejas como se quiera.

En ocasiones no hace falta que las operaciones sean muy complejas para llegar a perderse a la hora de saber qué resultado dará. Por ejemplo, esta operación:

```
i = - 3 + 4 * 5
```

Se puede interpretar de las siguientes maneras:

```
i = -((3 + 4) * 5) → i = -35
i = -(3 + (4 * 5)) → i = -23
i = -(3 + 4) * 5 → i = -35
i = (-3 + 4) * 5 → i = 5
i = -3 + (4 * 5) → i = 17
```


Como podéis ver, en un ejemplo sencillo en el que intervienen tres valores y tres operandos nos han surgido 5 interpretaciones posibles de la misma fórmula matemática. Por lo tanto, es necesario que el lenguaje defina un método para deshacer la ambigüedad de estas fórmulas (sin la necesidad de utilizar paréntesis).

El uso de los paréntesis

Siempre podemos utilizar los paréntesis para deshacer la ambigüedad de cualquier expresión.

En el lenguaje de programación Java, como en otros lenguajes, este método viene dado por el orden de precedencia de los operadores. Algunos operadores

tienen más precedencia que otros y, por lo tanto, se aplican primero. En la siguiente tabla tenemos un resumen de las precedencias:

| Típos | Operadores |  + precedencia |
|-----------------|---|---|
| Asignación | =, +=, -=, *=, /=, %=>, <=>, &=, =, ^= | |
| Unitarios | ++, --, &, ~, !, - | |
| Multiplicación | %, *, / | |
| Aditivos | +, - | |
| Desplazamientos | <<, >> | |
| Relacionales | <, <=, >, >= | |
| | ==, != | |
| Bit a bit | & | |
| | ^ | |
| | | |
| Lógicos | && | - precedencia |
| | | |
| Ternario | ? : | |

Esta precedencia se puede modificar siempre utilizando los paréntesis, como en el ejemplo anterior, agrupando los operandos con su operador.

Una característica de Java, al igual que muchos otros lenguajes de programación, es que no especifica en ningún momento el orden de ejecución de las operaciones dentro de la misma precedencia. Por lo tanto, suponiendo que el resultado de una operación dependiera de otra, es aconsejable realizar dos operaciones diferentes para evitar obtener resultados no esperados.

En el siguiente ejemplo no podemos asegurar si se evaluará primero el valor del preincremento y después se calculará la multiplicación, o si se hará en orden inverso:

```
i = 3;
i = ++i + i * j;
```

Para evitar posibles problemas es aconsejable, pues, escribir el código de la siguiente manera:


```
i = 3;
i_aux = i * j;
i = ++i + i_aux;
```

1.4. Matriz y vectores

Un **array** es una colección de elementos del mismo tipo que se identifican mediante un índice. Los **arrays** de un solo índice también se denominan **vectores**. También puede haber **arrays multidimensionales** (denominados **matrices**), es decir, que en vez de acceder al elemento con un índice, necesitamos una colección de valores (**tupla**) que indexen la posición pedida.

Para definir una matriz en Java, hay que indicar el tipo de datos que se almacenarán, el nombre con el que se identificará y el número máximo de elementos que se quieren almacenar. Si queremos definir una matriz que se denomine `days` para almacenar un número para cada día de la semana, lo haríamos de la siguiente manera:

```
int j;
int days[7];
...
days[3] = 5;
j = days[5];
```

Hay que tener en cuenta que en Java, como en otros lenguajes de programación, las matrices empiezan en la posición 0. 

La matriz anterior tendría los siguientes elementos:

```

      days[0]
      days[1]
      days[2]
      days[3]
      days[4]
      days[5]
      days[6]
int days[7] = [ ][ ][ ][ ][ ][ ][ ]
```

Para definir matrices (**arrays** multidimensionales) hay que definir la capacidad de cada dimensión. El sistema para acceder a cada celda es el mismo que en el caso anterior, sólo que ahora deberemos especificar N índices.

```
type name[][]...[] = new type[size_d1][size_d2]...[size_dn];
```

Para entender la utilidad de las matrices podemos pensar en un administrador de una página web que quiera almacenar el número de visitas que acceden a su página durante una semana. Para almacenar los datos, el administrador puede definir una matriz de 3 dimensiones donde la primera dimensión le puede servir para identificar el día de la semana, la segunda dimensión para la hora de la visita y la tercera dimensión para los minutos. Entonces, para regis-

trar un acceso el martes a las 15 horas y 24 minutos, deberíamos acceder de la siguiente manera:

```
int visits[][][] = new int[7][24][60];  
...  
visits[1][14][23]++;
```

1.5. Bloques de instrucciones

En el lenguaje Java, un **bloque de instrucciones** es una serie de órdenes separadas por un punto y coma que se suelen escribir en líneas diferentes con el fin de mejorar su legibilidad.

Los bloques de instrucciones se pueden agrupar utilizando las llaves '{' y '}' de modo que las instrucciones que están situadas dentro de las llaves se comportan como si fueran una única sentencia.

Los bloques de instrucciones se pueden clasificar principalmente en dos grupos: bloques condicionales y bloques iterativos, que pasamos a ver a continuación.

1.5.1. Bloques condicionales

Los **bloques condicionales** son aquellos que sirven para tomar decisiones y, según el resultado de éstas, ejecutar una serie u otra de instrucciones.

En el lenguaje Java, hay tres estructuras sintácticas que nos permiten escribir sentencias condicionales: los bloques `if-else`, `else-if` y `switch`. En realidad, las tres estructuras son equivalentes, es decir, las dos últimas no dejan de ser una manera más cómoda de expresar una composición de sentencias `if-else`, pero proporcionan más claridad al código.

1) Bloque `if-else`

El bloque `if-else` sirve para denotar dos bloques de instrucciones que se deben ejecutar de manera excluyente, según la evaluación de una expresión lógica. Por ejemplo, podemos expresar términos como “si es el lunes hacer... en caso contrario...”.

La **sintaxis** de este bloque de instrucciones es:

```
if (expression) {  
    // true expression  
}  
else {  
    // false expression  
}
```

Las expresiones que utilizamos han de generar un valor booleano.

Es obligatorio que la expresión esté delimitada por paréntesis, ya que forma parte de la sintaxis del lenguaje Java. En cambio, las llaves son elementos opcionales que sólo sirven para delimitar un bloque de instrucciones que se deben ejecutar según el resultado de la expresión.

Recomendamos...

... utilizar siempre llaves para hacer más inteligible el código.

Dependiendo de cuál sea el algoritmo, este bloque de instrucciones quizá no es necesario. Por lo tanto, podemos escribir sólo el código que pertenece al caso en que la expresión es cierta, y entonces la sintaxis de la construcción queda de esta manera:

```
if (expression) {  
    // true expression  
}
```


2) Bloque **else-if**

El bloque anterior nos sirve si queremos discriminar entre el valor 'cierto' y 'falso' de una expresión, pero resulta incómodo si queremos ejecutar un bloque de instrucciones diferente según el valor de diversas condiciones, como por ejemplo, expresar condiciones del tipo "si es lunes hacer... si es martes... si es miércoles...".

La **sintaxis** de esta construcción es la siguiente:

```
if (expression1) {  
    // true expression1  
}  
else if (expression2) {  
    // true expression2  
}  
else if (expression3) {  
    // true expression3  
}  
...  
else {  
    // false all expressions  
}
```

Las expresiones que utilizamos han de generar un valor booleano.

Si nos fijamos un poco, veremos que esta construcción no deja de ser una construcción **if-else** en la que los segundos bloques de acciones (los del **else**) no tienen claves. 

3) Bloque **switch**

El bloque `switch` nos ayuda a escribir condiciones en las que queremos evaluar una expresión y realizar una serie de operaciones u otras según el resultado. En realidad, estas condiciones se pueden escribir también con una sucesión de bloques `if-else`, pero resulta mucho más cómodo hacerlo con la construcción `switch`.

Veamos la **sintaxis** de esta construcción.

```
switch (expression) {  
    case (value1) :  
        // Code for value1  
        break;  
    case (value2) :  
        // Code for value2  
        break;  
    ...  
    default :  
        // Code for other values  
}
```

La expresión tiene que devolver un valor de tipo `char` o cualquier tipo de datos que se pueda convertir en un entero sin perder precisión (`byte`, `short`, `int`), que se compara con los valores de las cláusulas `case` y se ejecuta el código de la cláusula que corresponda. Todos los valores de las cláusulas `case` deben ser diferentes. En caso de que no se cumpla ninguna de las cláusulas, se ejecutaría el código de la cláusula `default`, que se considera la acción por defecto, en caso de que la haya. Si esta cláusula no existe, no se ejecutará ningún bloque y se continuará con la instrucción siguiente.

if y switch

Al contrario que la instrucción `if`, en el bloque `switch` la expresión no se evalúa a 'cierto' o 'falso'.

Una vez que se haya encontrado un valor que sea igual al resultado de la expresión, se ejecutará el código hasta encontrar la sentencia `break`. Entonces se sale de la ejecución del bloque condicional y se continúa a partir de la instrucción siguiente. Si queremos ejecutar un mismo bloque de instrucciones para diferentes valores, habrá que escribir las instrucciones `case` una detrás de la otra sin la palabra clave `break`.

```
switch (expression) {  
    case (value1) :  
        // Code for value1  
    case (value2) :  
        // Code for value1 & value2  
        break;  
    ...  
    default :  
        // Code for other values  
}
```

1.5.2. Bloques iterativos

Los **bloques iterativos** sirven para ejecutar un conjunto de instrucciones más de una vez.

En el lenguaje Java tenemos tres bloques iterativos diferentes: `while`, `do-while` y `for`.

Las tres estructuras sirven para lo mismo, pero cada una tiene unas características diferentes. Las veremos con más detenimiento a continuación.

1) Bloque `while`

El bloque `while` es el bloque iterativo más genérico. Sirve para expresar cualquier secuencia de acciones que se repitan.

La **sintaxis** de este bloque es la siguiente:

```
while (condition) {  
    // actions  
}
```

Las condiciones que utilizamos han de generar un valor booleano.

Para entenderlo mejor, veamos cuál sería el diagrama de ejecución de un bloque `while`.

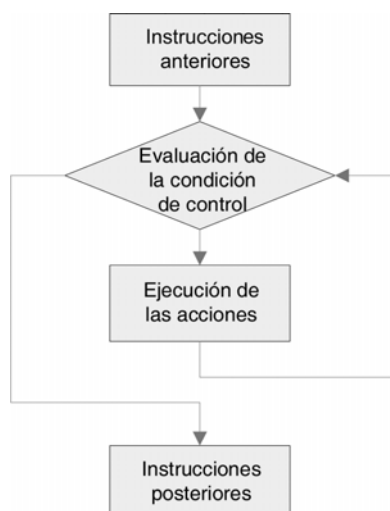


Figura 1. Diagrama de la sintaxis de un bloque `while`.

El bloque `while` funciona de modo que las instrucciones dentro de este bloque se ejecutan de manera continuada mientras la condición sea cierta. Cuando la evaluación de la condición es falsa, entonces se pasará a ejecutar las instrucciones que hay a continuación del bloque iterativo. La condición se evalúa antes de cada iteración.

¿Sabíais que...?

Para crear un bucle infinito basta sólo con escribir:
`while (true)`

Hay que ir con mucho cuidado con el fin de no codificar bucles infinitos, es decir, trozos de código que no dejan nunca ejecutarse. Los bucles infinitos son producidos principalmente por funciones de cota mal elegidas.

Funciones de cota

Una función de cota es aquella que utilizamos para asegurarnos de que un bloque iterativo se ejecutará siempre un número fijo de veces.

2) Bloque `do-while`

Otro tipo de bloque iterativo es el bloque `do-while`, que funciona de manera similar al bloque `while`, pero con la diferencia de que la condición se evalúa después de la ejecución del bloque de instrucciones.

La **sintaxis** de este bloque de instrucciones es la siguiente:

```
do {  
    // actions  
} while (condition);
```

En este caso también pueden aparecer bucles infinitos. Por lo tanto, hay que tener cuidado al definir las funciones de cota.

3) Bloque `for`

Finalmente, para terminar con los bloques iterativos, tenemos el bloque `for`. Utilizaremos esta construcción principalmente cuando queramos repetir ciertas instrucciones un número determinado de veces, por ejemplo “recorrer los elementos de una matriz”.

La sintaxis que siguen los bloques iterativos que se construyen con la sentencia `for` es la siguiente:

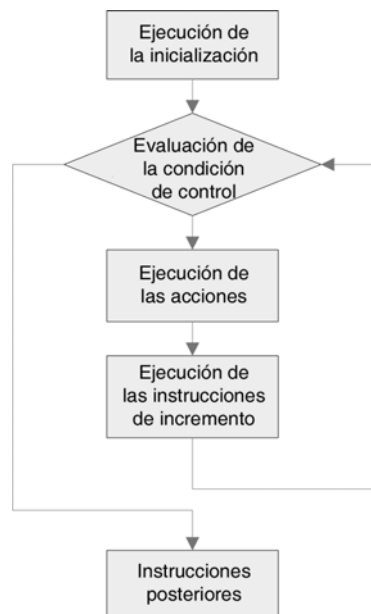
```
for (initialization; control condition; increment) {  
    // actions  
}
```

Esta construcción tiene un apartado en el que se inicializa la ejecución, normalmente dando valores a las variables que nos servirán para controlar el número de iteraciones, un segundo apartado donde ponemos la condición de ejecución del bucle y, finalmente, el tercer apartado, que se utiliza para incrementar las variables.

A continuación tenéis un diagrama que os ayudará a entender mejor el funcionamiento de esta construcción.

¿Sabíais que...?

Para construir un bucle infinito basta sólo con escribir:
`for (;;)`

Figura 2. Diagrama de la sintaxis de un bloque `for`.

Como hemos visto, el bloque `for` está compuesto por tres partes. Estas partes pueden estar vacías, pero recomendamos que siempre se pongan las sentencias de inicialización, condición de control e incremento con el fin de hacer más inteligible el código y evitar posibles problemas en la ejecución.

1.5.3. Sentencia `break`

La sentencia `break` sirve para interrumpir la ejecución de un bloque iterativo en cualquier momento.

Recordad

La sentencia `break` también se utiliza en el bloque `switch`.

Aunque el lenguaje Java permite esta posibilidad, no es en absoluto recomendable, ya que hace difícil la comprensión del código y siempre se puede escribir el código sin hacer uso de esta sentencia.

1.5.4. Sentencia `continue`

La sentencia `continue` también sirve para alterar el funcionamiento de los bloques iterativos, pero en vez de interrumpir la ejecución del bloque iterativo, se inicia una iteración nueva.

Con la sentencia `continue`, pues, la condición se vuelve a evaluar y en el caso de la construcción `for` se ejecutan también las instrucciones de incremento de las variables.

Así como desaconsejamos el uso de la sentencia `break`, desaconsejamos también el uso de la sentencia `continue`, ya que se puede conseguir la misma funcionalidad sin penalizar la legibilidad del código.

1.6. Funciones

Todos los conceptos que hemos visto hasta ahora nos han permitido tener una visión profunda del lenguaje de programación Java para poder programar aplicaciones sencillas que utilizan bloques iterativos, bloques condicionales y operaciones matemáticas.

Un concepto muy importante en la programación estructurada es el concepto de *función*.

Una **función** es un trozo de código que ejecuta una tarea determinada. Esta tarea está constituida por una serie de instrucciones.

¿Sabíais que...?

Normalmente, se utiliza el nombre de *función*, pero también se les puede denominar, *subrutinas*, *procedimientos*, *subprogramas*, y en programación orientada a objetos, *métodos*.

Las funciones evitan al programador tener que repetir el mismo código varias veces y hacen más inteligible el funcionamiento de un programa. Por ejemplo, podríamos tener una función que calculara la media aritmética de los valores de un vector de enteros, o el valor máximo, etc. (cualquier tarea que se nos ocurra que hubiéramos de utilizar varias veces dentro de nuestro programa).

1.6.1. Definición de funciones

Para poder utilizar algo tan útil para el programador como una función, antes hay que definirla. La instrucción que define una función también se denomina **cabecera** de la función.

La **sintaxis** que corresponde a la cabecera de una función es la siguiente:

```
ReturnDataType FunctionName(Parameters) {  
    // Code to be executed  
}
```

En este caso no le hemos añadido el concepto de *visibilidad*. Podéis repasar este concepto en el módulo "Abstracción y clasificación" de esta asignatura.

Lo primero que deberemos decidir para una función es su nombre. Las funciones, igual que las variables, siguen las mismas convenciones de nomenclatura. Así pues, intentaremos ponerles un nombre esclarecedor que nos permita intuir qué tarea ejecuta la función. Por ejemplo, para nombrar una función que calcule el máximo de dos números enteros podemos utilizar `max`, `int_max`, o cualquier otro nombre, pero es aconsejable utilizar un nombre bastante claro (no sería adecuado utilizar como nombre `min`).

Podéis ver las mismas convenciones de nomenclatura en Java en el subapartado 1.1.4 de este módulo.

Otros puntos que habrá que decidir para definir una función son el valor que devuelve y los parámetros que recibe, lo cual veremos en los subapartados siguientes.

1.6.2. Los parámetros de entrada

En la mayoría de los casos habrá que facilitar a la función un conjunto de valores sobre los que queremos que ejecute las operaciones que le deben permitir llevar a cabo su tarea.

En la definición de las funciones debemos indicar qué parámetros recibirá la función, el tipo de parámetros y su orden. La manera de hacerlo es mediante una lista. Por ejemplo, la definición de la función que nos calcula el máximo de dos números enteros sería:

```
int max(int a, int b) {
```

Como podéis observar, no sólo se pone el tipo del parámetro, sino también un nombre identificativo del parámetro. Estos nombres son los que deberemos utilizar dentro del código de esta función para referirnos a los parámetros, como si se tratara de variables declaradas normalmente. Estos nombres únicamente nos servirán dentro de la función.

1.6.3. El valor de retorno

Una función suele realizar un cálculo o cualquier tarea que devuelve un resultado. Hay que indicar de alguna manera cuál es el tipo de datos que devuelve.

Como hemos visto en la definición de funciones, el tipo de datos que será devuelto se indica antes de especificar el nombre que tendrá y éste puede ser cualquier tipo de datos de entre los que hemos explicado: `int`, `double`, `char`, etc., o un objeto de una clase.

En el ejemplo del punto anterior, la función `max` devuelve un valor de tipo `int`.

Para devolver un valor es necesario utilizar la sentencia `return`, que tiene el formato siguiente:

```
return expression;
```

Aquí, `expression` puede ser cualquier expresión que genere un valor del tipo definido en la cabecera de la función. Tan pronto se ejecuta una sentencia `return`, se retorna al punto del código donde la función ha sido llamada.

Hay funciones que no devuelven ningún valor, ya sea porque modifican variables globales, ya sea porque llevan a cabo tareas como la escritura en un archivo y/o por pantalla. Estas funciones se pueden declarar de la misma manera que las otras, únicamente hay que indicar que devuelven un elemento de tipo `void`.

1.6.4. Un ejemplo de función

Para aclarar las ideas sobre las funciones, veamos cómo sería una función que calcula el máximo de dos enteros que se pasan como parámetro.

```
public int max (int a, int b) {  
    int aux;  
    if (a < b) {  
        aux = b;  
    }  
    else {  
        aux = a;  
    }  
    return aux;  
}
```

Esta función recibe dos parámetros de entrada, *a* y *b*, los dos enteros, y devuelve el valor máximo.

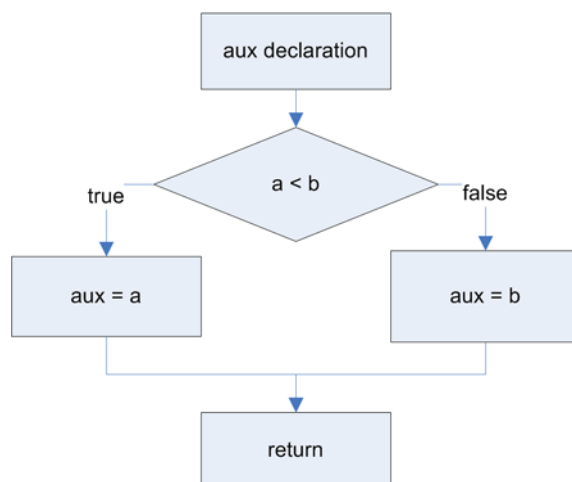


Figura 3. Diagrama de flujo de la función `max`

La función determina cuál es el valor máximo mediante un bloque condicional `if-else`. Utiliza una variable auxiliar llamada `aux` para almacenar el resultado, y finalmente devuelve el valor de esta variable auxiliar.

1.6.5. La invocación de funciones

Ya hemos aprendido a definir y llenar de contenido una función. Ahora debemos saber cómo hacer la llamada, o invocación, para obtener el resultado. De hecho, no hay que hacer nada especial, únicamente escribir el nombre de la función junto con los parámetros de ésta en el instante en que se deba ejecutar.

Podríamos decir que una **llamada a una función** es exactamente igual que evaluar una expresión que diera como resultado el valor devuelto por la función.

Asignemos el resultado de la invocación a una variable:

```
int i;  
i = max(3, 4);
```

Utilizamos el resultado como operando de una expresión:

```
if (8 < max(9, 3)) {  
    ...  
}
```

O simplemente no recogemos el valor devuelto:

```
System.out.println("I'm writting this on the screen\n");
```

El método `println` escribe para la salida estándar (normalmente la pantalla) la cadena dada.

1.7. Visibilidad de las variables

Hasta ahora no nos hemos preocupado de ver si podíamos acceder o no al valor de las variables, siempre hemos utilizado variables que declarábamos justo antes de usarlas y, por lo tanto, siempre eran accesibles. Pero eso no siempre es así.

Para entender la visibilidad de las variables hay que recordar un concepto ya explicado: el bloque de instrucciones. Recordemos que un bloque de instrucciones es un conjunto de sentencias que están delimitadas por un { y un }.

Definimos la **visibilidad de una variable** (también denominada **ámbito de una variable**), como la región de un programa desde donde se puede hacer referencia a esta variable por su nombre y acceder a su valor.

Según el tipo de variable y, principalmente, según su visibilidad, podemos tener variables locales y variables globales, que pasamos a explicar a continuación.

1.7.1. Variables locales

Las **variables locales** son las que sólo son accesibles desde dentro de un bloque de instrucciones.

A continuación tenemos un ejemplo de una posible implementación de la función `max`.


```
int max(int a, int b) {  
    if(a > b) {  
        int aux = 1;  
    }  
    else {  
        int aux = 0;  
    }  
    return aux; // Compilation error  
}
```

En el ejemplo anterior, el compilador nos dará un error porque en la sentencia `return` no es visible ninguna variable con nombre `aux`. Las variables `aux` dejan de ser visibles con el `}`. Para solucionar este problema hay que escribir la función de esta manera:

```
int max (int a, int b) {  
    int aux;  
    if (a < b) {  
        aux = b;  
    }  
    else {  
        aux = b;  
    }  
    return aux;  
}
```

De este modo se amplía la visibilidad de la variable `aux`, que ya no nos dará ningún error de compilación.

De la misma manera que no podemos acceder a variables que están fuera de nuestro ámbito de visibilidad, tampoco podemos acceder a variables que se encuentran en otras funciones.

```
int max(int a, int b) {  
    int res = 0;  
    ...  
    return res;  
}  
  
int min(int a, int b) {  
    if (a > b) {  
        res = 0; // Compilation error. Variable res not defined  
    }  
    ...  
    return res;  
}
```

1.7.2. Variables globales

Las **variables globales** son aquellas que son accesibles desde cualquier punto de la ejecución del programa, incluso desde dentro de funciones diferentes.

Normalmente, la presencia de variables globales en un programa suele denotar un mal diseño de las aplicaciones.

2. Java como lenguaje de programación orientado a objetos

Como hemos visto, Java se puede utilizar para crear programas utilizando el paradigma de la programación estructurada pero, en realidad, donde se consigue aprovechar al máximo la potencia del lenguaje es con la programación orientada a objetos. En este apartado veremos las posibilidades que nos ofrece Java para el trabajo utilizando clases, concepto primordial de la programación orientada a objetos. Primero veremos cómo se define una clase, sus métodos y atributos. A continuación, cómo esta clase se relaciona con otras clases con el fin de modelar la realidad necesaria y resolver el problema planteado. Después, trataremos algunos temas avanzados para no quedarnos en la superficie, pero dado que esto no es un manual de Java, sino sólo una introducción para poder aprender los conceptos de la programación orientada a objetos, otros temas como la explicación exhaustiva de la API de Java, o el diseño de interfaces gráficas quedarán excluidos del presente manual.

2.1. Definición de clases

Para empezar a trabajar con programación orientada a objetos es necesario que aprendamos a definir una clase con sus atributos y métodos. Más adelante veremos cómo representamos en Java las relaciones entre clases y la invocación de métodos de otras clases.

2.1.1. La clase

Para describir una clase en Java hay que definir y codificar todos los métodos y atributos que la componen. Crearemos un archivo con extensión `.java` en el que escribiremos tanto la definición de la clase, como su código (en Java no hay otra manera de hacerlo). Veamos, en primer lugar, cuál es la estructura del archivo de una clase y después ya veremos cómo la llenamos de contenido.

MyFirstClass.java

```
public class MyFirstClass {  
    private int myVar;  
  
    public int myPublicMethod(int oneVar) {  
        // some code here  
    }  
  
    private int myPrivateMethod(int anotherVar) {  
        // some more code  
    }  
}
```

Recordad

En Java, la clase `Person` tiene que estar en el archivo `Person.java`; si no lo hacemos así, el compilador dará un error.

La clase se define como `public` para que se pueda utilizar desde otros programas.

Por una parte, se denotan los atributos y posteriormente las firmas de los métodos, teniendo en cuenta su visibilidad. Aunque esta estructura no está impuesta por el lenguaje, es una buena práctica para ganar en claridad en el código y permitir una mejor lectura a terceras personas.

Para recordar el concepto de visibilidad, podéis repasar el módulo "Abstracción y clasificación" de esta asignatura.

Para mejorar nuestro código se recomienda que todos los atributos sean privados y que sólo se expongan mediante métodos consultores y/o modificadores aquellos que sean imprescindibles; de esta manera utilizamos el concepto de ocultación de información y el encapsulamiento.

Para declarar un atributo, basta describir cuál será su visibilidad, seguida del tipo (puede ser tanto un tipo de dato básico como cualquier objeto) y posteriormente su nombre.

Para declarar los métodos, se escribe primero la visibilidad, seguida de su firma, es decir, el valor que devuelve, el nombre del método y sus parámetros. Una vez definida la signatura (o firma), escribiremos entre llaves el código correspondiente al método.

2.1.2. Los atributos de una clase

Los atributos de una clase se definen utilizando la siguiente construcción:

```
visibilityMode type name;
```

Donde, como ya hemos comentado antes, hay que indicar inicialmente la visibilidad que tendrá el atributo, posteriormente el tipo y a continuación el nombre de éste (siguiendo las convenciones del subapartado 1.1.3 de este módulo).

Veamos cómo definiríamos un atributo de la clase `Person`, en este caso, el nombre de la persona.

`Person.java`

```
public class Person {  
    private String name;  
}
```

Se pueden definir atributos y métodos como `public`, `private` y `protected`. Estos modificadores nos permiten indicar la visibilidad tanto de los métodos, como de los atributos.

Del mismo modo, si queremos definir un atributo de clase, es decir, un atributo que tenga el mismo valor para cualquiera de las instancias de la clase, hay que indicar el modificador `static` delante de su tipo.

Los atributos de clase

Un atributo de clase puede servirnos, por ejemplo, para poder contabilizar el número de instancias de una clase que hemos creado, etc.

2.1.3. Métodos de una clase

Llegados a este punto, ya somos capaces de crear una clase con sus atributos. Nuestro objetivo es tener una clase plenamente funcional con la que podamos interaccionar invocando métodos tanto para consultar datos de ésta como para modificar su estado.

Veamos cómo se debe llevar a cabo con el fin de poder definir métodos en una clase.

2.1.4. El método constructor

El **método constructor** es el que se utiliza para crear nuevas instancias de aquella clase.

Se puede definir más de un método constructor. Podéis ver la sobrecarga de métodos en el subapartado 2.1.9 de este módulo.

El método constructor debe tener el mismo nombre que la clase que se define, pero puede tener cualquier número de parámetros, según se requiera.

A continuación, tenemos un ejemplo de la clase `Person` con un método constructor que únicamente asigna a un atributo la cadena de texto pasada como parámetro.

`Person.java`

```
public class Person {  
    private String name;  
  
    public Person(String pName) {  
        name = pName;  
    }  
}
```

Como podemos ver, el método constructor no devuelve ningún valor. Esto se debe a que, en realidad, devuelve una instancia de la clase. !

Dentro del método constructor se llevan a cabo las principales tareas de inicialización de los elementos de la instancia de la clase. Aunque se pueden realizar mediante llamadas a otros métodos, es recomendable inicializar todo en el momento de la creación del objeto para prevenir errores.

El método constructor, como cualquier otro método, permite la sobrecarga; esto significa que podemos tener definidos varios métodos constructores con diferentes parámetros.

Podéis ver la sobrecarga de métodos en el subapartado 2.1.9 de este módulo.

Además, el lenguaje de programación Java, en caso de que definiéramos una clase sin ningún método constructor, genera una de manera automática sin ningún parámetro y que no ejecuta ninguna tarea.

2.1.5. El método destructor

De la misma manera que tenemos un método para crear una instancia de una clase, hay un método para destruir una instancia de una clase.

Este método es llamado de manera automática al perderse la visibilidad de una instancia de la clase. Por lo tanto, habrá que realizar las tareas necesarias para liberar los posibles recursos reservados (cerrar archivos, liberar memoria, conexiones en bases de datos, etc.).

El compilador de Java, por defecto, nos crea un método destructor. Aun así nos puede interesar crear uno nosotros mismos. El método destructor se denomina `finalize` y no recibe ningún parámetro.

La clase `Person` que hemos definido antes quedaría de la siguiente manera:

`Person.java`

```
public class Person {  
    private String name;  
  
    public Person(String pName) {  
        name = pName;  
    }  
  
    public void finalize() {  
        // do something to erase  
    }  
}
```

En este caso no hemos efectuado ninguna tarea dentro del método destructor, dado que en el método constructor no hemos pedido recursos, pero es algo que debemos tener en cuenta para crear aplicaciones correctas.

2.1.6. El resto de métodos

Los dos tipos de métodos anteriores nos han servido para iniciar una nueva instancia de la clase y para liberar recursos una vez que se deje de utilizar esta instancia. Ahora hemos de ver cómo definir el resto de métodos de la clase: los métodos consultores y los métodos modificadores.

Del mismo modo que hemos hecho con el método constructor y el destructor, hay que indicar la signatura de las operaciones: el nombre, el tipo del valor de retorno (en caso de que no devuelva ningún valor hay que indicarlo mediante la palabra `void`), y el nombre y tipo de cada parámetro.

Si seguimos con el ejemplo anterior de la clase `Person`, ahora deberemos implementar un par de métodos para recuperar y modificar el nombre de la clase. Estos métodos se definirían de la siguiente manera:

Pérdida de visibilidad de un objeto

Un objeto pierde su visibilidad cuando ya no es accesible utilizando el nombre de la variable que lo identificaba (podéis ver el subapartado 1.7 de este módulo).

Person.java

```
public class Person {
    private String name;

    public Person(String pName) {
        name = pName;
    }

    public void finalize() {
        // do something to erase
    }

    public String getName() {
        return name;
    }

    public void setName(String pName) {
        name = pName;
    }
}
```

En este caso, el método `getName` no recibe ningún parámetro y devuelve una cadena de texto y el método `setName` no devuelve ningún valor y recibe como parámetro una cadena de texto.

2.1.7. Uso de la palabra reservada `this`

En el ejemplo anterior hemos definido e implementado el método `setName` de la siguiente manera:


Person.java

```
...
    public void setName(String pName) {
        name = pName;
    }
...
```

Pero nada nos impide hacerlo de esta manera:

Person.java


```
...
    public void setName(String name) {
        name = name;
    }
...
```

Por lo tanto, tenemos un problema dentro del método `setName`, ya que no podemos diferenciar de manera clara el atributo `name` del parámetro `name`. Esta diferenciación se puede alcanzar mediante la palabra reservada `this`. 

Para evitar posibles ambigüedades, en la implementación de estos métodos hay que escribir la palabra `this` ante los atributos de la clase. La palabra clave `this` nos permite referirnos a la clase del objeto. El código resultante quedaría de esta manera:

Person.java

```
...  
    public void setName(String name) {  
        this.name = name;  
    }  
...
```

La palabra reservada `this` también sirve para denotar que el método que invocamos está definido dentro de la misma clase. 

2.1.8. Métodos estáticos

Los métodos que hemos definido hasta ahora retornaban o modificaban información en lo referente a la instancia de la clase sobre la que se invocaba el método. Ahora bien, nos puede interesar tener un método que no dependa del estado de la instancia sobre la que se ejecuta. Es decir, que devuelva un mismo resultado para unos mismos parámetros, sin importar sobre qué instancia concreta se invoque. Estos métodos se denominan métodos `static`.

Un ejemplo de esto podría ser un método que, dada una fecha, sumara o restara una cierta cantidad de días. Lo normal sería implementarlo en una clase `Date` y definirlo de esta manera:

Data.java

```
public class Date {  
    ...  
    public static Date addDates(Date pDate, int days) {  
        // some code here  
    }  
}
```

Accesibilidad de los métodos

Un método estático sólo puede acceder a atributos estáticos; en cambio, un método no estático puede acceder a atributos tanto estáticos como no estáticos.

2.1.9. Sobrecarga de métodos

En este momento ya tenemos una clase creada, con atributos y métodos. Sin embargo, con lo que hemos explicado hasta ahora, tenemos la limitación de que sólo podemos tener un método con el mismo nombre. Esto representaría tener que definir diferentes métodos para la misma tarea con nombres diferentes, por ejemplo, tener una clase que se encarga de calcular operaciones matemáticas y querer definir una operación para sumar dos números.

Por lo tanto, tendríamos las siguientes definiciones de métodos:

MathOperation.java

```
public class MathOperation {  
    public static int add_int_int(int a, int b) {...}  
    public static int add_int_long(int a, long b) {...}  
    public static int add_int_float(int a, float b) {...}  
    public static int add_int_int_int(int a, int b, int c) {...}  
    ...  
}
```

Aunque esta manera de trabajar no es incorrecta, tampoco es la mejor, ya que la nomenclatura se vuelve farragosa. Es más aconsejable utilizar la sobrecarga de métodos.

La **sobrecarga de métodos** nos permite definir diferentes métodos con el mismo nombre dentro de una misma clase, siempre y cuando el número y el orden de los tipos de los parámetros sea diferente.

Con este concepto, la manera más correcta de definir la clase anterior sería ésta:

MathOperation.java

```
public class MathOperation {  
    public static int add(int a, int b) {...}  
    public static int add(int a, long b) {...}  
    public static int add(int a, float b) {...}  
    public static int add(int a, int b, int c) {...}  
    ...  
}
```

Así, el usuario sabe que esta clase le permite calcular sumas de números y que el método utilizado siempre se denomina `add`.

2.2. Las relaciones entre clases

En este subapartado comentaremos los principios básicos para implementar las relaciones entre clases, que ya hemos visto en cuanto al diseño.

Podéis ver las relaciones entre clases en el módulo "Abstracción y clasificación" de esta asignatura.

2.2.1. Cardinalidad

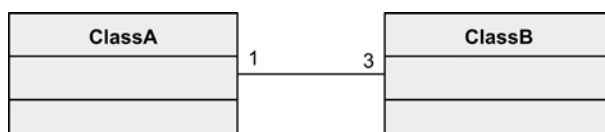
Como ya hemos mencionado, una relación siempre tiene una cardinalidad asociada que nos indica el número de instancias con las que puede estar relacionada.

Para representar la cardinalidad de las relaciones de la programación orientada a objetos en Java, hay que tener una referencia a objetos en nuestra clase en forma de un atributo que representará esta relación.

Dado que hay diferentes tipos de cardinalidad (número exacto, rango de valores y cardinalidades indefinidas), son necesarias maneras diferentes para cada tipo. Veamos cómo se implementa cada una de estas tipologías.

2.2.2. Nombre exacto

Consideramos el diagrama de clases siguiente:



Observamos que una instancia de `ClassB` siempre está relacionada con una (1) única instancia de `ClassA`. Por lo tanto, definiríamos en `ClassB` el siguiente atributo del tipo `ClassA` para representar esta relación:

`ClassB.java`

```
public class ClassB {
    ClassA a1;
    ...
}
```

Para implementar completamente la relación habría que definir tres atributos del tipo `ClassA` que representarían el otro lado de la relación. Esta manera de trabajar nos plantea un grave problema: ¿qué hacemos si tenemos una cardinalidad muy grande? No resulta práctico definir 100 atributos con nombres similares, ya que estos nombres resultan poco intuitivos para el programador.

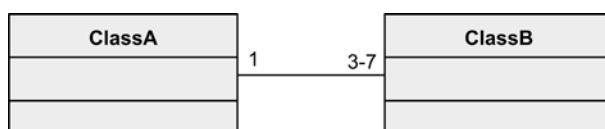
En este caso habría que crear un vector con capacidad para el número de instancias con las que estará relacionado e insertar cada una de las instancias en una posición de éste.

Otra posibilidad

Otra manera de solucionar este problema es utilizando la solución que propondremos para los valores indefinidos (subapartado 2.2.4).

2.2.3. Rango de valores

En caso de que tengamos un rango de valores como en el siguiente diagrama:



Nos encontramos con el mismo problema que teníamos en el caso anterior, por el hecho de tener una cardinalidad superior a 1. Definir un atributo para

cada instancia es factible sólo para pocos valores. Es más común definir un vector que nos permita almacenar todas las instancias.

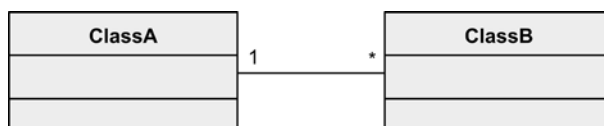
Así, en el ejemplo anterior definiríamos un vector de 7 posiciones en `ClassA` de elementos de `ClassB`. Dado que tenemos un mínimo de 3 instancias relacionadas y un máximo de 7, si definimos el vector de tamaño inferior, nos podríamos encontrar sin suficiente espacio para almacenar los datos correspondientes. Siempre definiremos el valor máximo dentro del rango.

2.2.4. Valores indefinidos

En caso de que tuviéramos un número indefinido de valores (una relación del tipo `*`) en vez de un rango predefinido, la solución óptima (aunque posible) no es un vector, ya que cada vez que insertáramos un elemento y excediéramos su capacidad, deberíamos volver a crear un vector nuevo de más capacidad y copiar en él los elementos. Hemos de buscar otro modo de almacenar estas instancias.

Para hacerlo, habrá que usar clases auxiliares que permitan un crecimiento dinámico del número de elementos del vector. Con esta intención, podemos utilizar cualquiera de las clases contenedoras que nos ofrece el API de Java. En nuestro caso proponemos utilizar `ArrayList`, pero en otros casos y según las necesidades que tengamos en la busca de instancias de la relación podremos utilizar otras clases.

Un ejemplo de cómo quedaría `ClassA` del siguiente diagrama sería:



`ClassA.java`

```
import java.util.ArrayList;

class ClassA {
    private ArrayList<ClassB> v;
    ...
}
```

! Tenéis información sobre listas y vectores en el módulo "Estructuras de objetos" de esta asignatura.

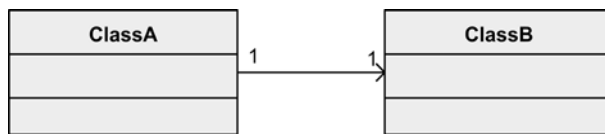
La clase `ArrayList`

La clase `ArrayList` es una clase parametrizada, lo que nos permite indicar en tiempos de compilación el tipo de objetos que almacenará.

2.2.5. Navegabilidad

La **navegabilidad**, como ya sabemos, denota la conciencia, por parte de una clase, de la propia relación. Hablando en términos de implementación, este concepto nos indica en cuál de las clases hemos de definir el atributo, la matriz o el vector, dada la multiplicidad de la relación.

En navegabilidades unidireccionales debemos declarar el atributo en la clase de la que sale la flecha indicativa. En este caso, el atributo debe estar declarado en `ClassA`. !



En navegabilidades bidireccionales, el atributo se debe declarar en las dos clases. !

2.2.6. Roles

El rol de las clases en las asociaciones sólo nos indica el nombre del atributo que utilizaremos.

Aunque los atributos pueden tener el nombre que nos parezca más adecuado, es recomendable utilizar el nombre del rol con el fin de mantener una cierta coherencia con el diagrama UML.

2.3. Biblioteca de clases

En Java, como en otros lenguajes de programación orientados a objetos, se dispone de una serie de clases ya implementadas. Este conjunto de clases se denomina *biblioteca de clases*.

Hay muchas bibliotecas de clases diferentes que permiten realizar desde tareas sencillas hasta tareas muy complejas, como la encriptación, el tratamiento de imágenes, el trabajo con interfaces gráficas, etc.

En Java, tenemos un conjunto de clases que conforman el **API de Java** y que nos ofrecen todas estas utilidades. Como este conjunto de clases es muy grande y está en constante evolución, debido a las mejoras que se van añadiendo poco a poco, únicamente comentaremos, y muy por encima (ya que si queréis más información, siempre tenéis la ayuda oficial), un par de clases que utilizaremos a lo largo de todos los materiales.

API es la sigla del término inglés *Application Programming Interface*.

En primer lugar, veremos cómo utilizar la clase `String`, que nos sirve para tratar cadenas de caracteres, y a continuación daremos también una ojeada a la clase `ArrayList`, que nos permite almacenar objetos de cualquier tipo.

! Para completar la información sobre cualquier clase de la API de Java, podéis visitar la siguiente página web: <http://java.sun.com/reference/api/>

2.3.1. La clase String

La clase **String** representa una secuencia de caracteres y ofrece una serie de operaciones que permiten trabajar de manera sencilla y cómoda.

Para poder trabajar con objetos de la clase `String`, no hay que hacer nada especial, ya que esta clase se encuentra dentro de la biblioteca `java.lang`, y esta biblioteca se utiliza por defecto en todos los programas Java. Por lo tanto, lo primero que debemos hacer es declarar un objeto de tipo `String` y, a continuación, crear el objeto. Esto se puede realizar de varias maneras, tantas como métodos constructores tenemos (13 en estos momentos) más la asignación de una cadena de caracteres delimitados entre comillas dobles que tiene la misma funcionalidad.

Algunos ejemplos de creación de objetos de tipo `String` son:

```
String s0 = "abc";
String s1 = new String();
char data[] = {'a', 'b', 'c'};
String str = new String(data);
String s3 = new String(s0);
```

Hay otros modos de crear e iniciar objetos `string`. Los encontraréis todos en la API.

Una vez ya tenemos un objeto de tipo `String` creado, podemos ejecutar muchas operaciones. Entre otras, hay operaciones de asignación, u operaciones que nos muestran la longitud de la cadena, que nos permiten efectuar búsquedas, etc.

Veamos algunas de las opciones en forma de operaciones que esta clase nos ofrece:

Observación

No es el objetivo de este apartado ver todas las posibilidades, pero hay que saber que existen sobrecargas de los métodos aquí presentados.

- `charAt`: proporciona el carácter de una posición determinada.
- `compareTo`: compara dos cadenas lexicográficamente.
- `compareToIgnoreCase`: compara dos cadenas lexicográficamente ignorando las mayúsculas y minúsculas.
- `concat`: concatena la cadena que representa objetos con el reparto.
- `contains`: indica si existe una cadena de caracteres concreta en el objeto.
- `endsWith`: comprueba si la cadena acaba con una secuencia concreta.
- `indexOf`: devuelve la posición de un carácter dentro del objeto.
- `lastIndexOf`: devuelve la última posición de un carácter dentro del objeto.
- `length`: devuelve la longitud de la cadena almacenada.
- `matches`: comprueba si la cadena cumple una expresión regular.
- `replace`: reemplaza una parte de una cadena por otra.
- `split`: parte una cadena utilizando una expresión regular.
- `startsWith`: comprueba si la cadena empieza de una determinada forma.
- `substring`: devuelve una subcadena de la cadena original.
- `toLowerCase`: transforma la cadena a minúsculas.
- `toUpperCase`: transforma la cadena a mayúsculas.

- `trim`: elimina los espacios sobrantes delante y detrás de la cadena.
- `valueOf`: devuelve la cadena que representa el objeto/variable dado.

Para ver cómo funciona la clase `String`, a continuación tenemos un trozo de código que utiliza muchas de las operaciones mencionadas.

TestString.java

```
public class TestString {

    public static void main(String[] args) {

        String s0 = "Something inside";
        String s1 = "GUAU!";
        String s2 = "The dog says ... ";
        String s3 = new String("The cat says... ");
        String s4 = new String("MIAU!");

        System.out.println("The lowercase value of s0 is : " + s0.toLowerCase());
        System.out.println("The uppercase value of s0 is : " + s0.toUpperCase());
        System.out.println("The value of s1 is : " + s1 + " and has " + s1.length() +
            " characters");
        System.out.println("The value of s2 is : " + s2 + " and has it's first 's' on " +
            s2.indexOf('s') + " position");
        System.out.println("The trimmed value of s3 is : \"" + s3.trim() + "\"");
        s2 = s2.concat(s1);
        s3 = s3.concat(s4);
        System.out.println("The concatenation of s2 and s4 is : " + s2);
        System.out.println("The first 5 characters of s3 are : \"" + s3.substring(0, 5) + "\"");
        if (s3.startsWith(s4)) {
            System.out.println("s3 starts with \"" + s4 + "\"");
        }
        else {
            System.out.println("s3 doesn't start with \"" + s4 + "\"");
        }
        if (s3.endsWith(s4)) {
            System.out.println("s3 ends with \"" + s4 + "\"");
        }
        else {
            System.out.println("s3 doesn't end with \"" + s4 + "\"");
        }
        if (s2.contains("dog")) {
            System.out.println("s3 contains the \"dog\" substring");
        }
        else {
            System.out.println("s3 doesn't contains the \"dog\" substring");
        }
        System.out.println("The value of s3 after a replacement : " + s3.replace(s4, s1));
        System.out.println("The value of a boolean false is : " + String.valueOf(false));
        System.out.println("The value of the max int value is : " +
            String.valueOf(Integer.MAX_VALUE));
        String splits[] = s2.split(" ");
        System.out.println("The content of splits is :");
        for(int i = 0; i < splits.length; i++) {
            System.out.println "[" + i + " ] : " + splits[i]);
        }
    }
}
```

Las operaciones que hemos utilizado tienen más de una sobrecarga, por lo tanto hay que echar un vistazo a la documentación de la clase `String` con el fin de conocer todas las opciones que esta clase nos ofrece.

2.3.2. La clase `ArrayList`

Como ya hemos visto, la clase `ArrayList` sirve para almacenar instancias de otros objetos y recuperarlas posteriormente para hacer lo que sea necesario. De la misma manera, cuando hemos comentado las relaciones entre clases en Java, también hemos introducido los `ArrayList`; por lo tanto, iremos directamente a cómo podemos hacer para trabajar con ella.

Primero hemos de declarar una variable de tipo `ArrayList`, como lo haríamos con cualquier otra variable, y a continuación deberemos llamar a uno de sus métodos constructores, como en el caso de la clase `String`. Por ejemplo, la declaración e inicialización posterior de un `ArrayList` de personas, tendría esta forma:

```
ArrayList<Person> persons;  
...  
persons = new ArrayList<Person>();
```

Ahora hay que ver cuáles son las operaciones que podemos realizar sobre un `ArrayList`. Dado que éste implementa la interfaz `List`, podremos realizar las mismas operaciones que ésta define, más las propias de la clase.

Del mismo modo como hemos hecho con la clase `String` veremos también un subconjunto de las operaciones que la clase `ArrayList` tiene definidas, ya que si queremos verlas todas, siempre podemos utilizar la API de Java.

- `add`: añade un elemento a la lista.
- `addAll`: añade una lista a la lista (equivalente a concatenar).
- `clear`: elimina todos los elementos de la lista.
- `clone`: clona la lista, los objetos internos no son clonados.
- `contains`: comprueba si un objeto existe en la lista
- `ensureCapacity`: incrementa la capacidad máxima de la lista.
- `get`: devuelve el elemento pedido de la lista.
- `indexOf`: busca la primera posición de un elemento dentro de la lista.
- `lastIndexOf`: busca la última posición de un elemento dentro de la lista.
- `remove`: elimina el elemento de la posición indicada.
- `removeRange`: elimina los elementos del rango dado.
- `set`: sustituye el elemento de la posición indicada con el objeto dado.
- `size`: indica la cantidad de elementos almacenados.
- `toArray`: devuelve un *array* con todos los elementos de la lista.
- `iterator`: ofrece un iterador con el fin de poder recurrir la lista en recorridos.

En el trozo de código siguiente tenéis unas cuantas de las operaciones que se pueden efectuar con `ArrayList`. Como esta clase almacena clases, utilizare-

Podéis ver la clase `ArrayList` en el módulo "Un ejemplo práctico" de esta asignatura. Podéis ver las relaciones entre clases en el subapartado 2.2 de este módulo.

Para conocer más métodos de la clase `ArrayList`, podéis consultar la API.

iterator

Un `iterator` es un objeto que nos ayuda a recorrer los elementos de un conjunto de manera más simple.

mos una clase `Person` que tenemos implementada en el archivo `Person.java`. Esta clase únicamente tiene un atributo de tipo `String` (el nombre de la persona), un método consultor y otro modificador.

Aquí no tenemos esta clase `Person` implementada, ya que no es el objetivo de este subapartado, pero llegados a este punto, no deberíais tener ningún problema para implementarla siguiendo los pasos de los subapartados anteriores.

`TestArrayList.cpp`

```
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

public class TestArrayList {
    public static void main(String[] args) {
        Person p1 = new Person("Luke");
        Person p2 = new Person("Leia");
        Person p3 = new Person("Anakin");
        Person p4 = new Person("Amidala");

        List<Person> persons = new ArrayList<Person>();
        List<Person> parents = new ArrayList<Person>();

        parents.add(p3);
        parents.add(p4);

        persons.add(p1);
        persons.add(p2);
        persons.addAll(parents);
        parents.clear();

        System.out.println("Persons list size : " + persons.size());
        System.out.println("Parents list size : " + parents.size());
        Person p = persons.get(2);
        System.out.println("Person's name : " + p.getName());

        if (parents.remove(p4)) {
            System.out.println("On the parents list there's only the dark side");
        }
        else {
            System.out.println("Something went wrong!!");
        }
        Iterator it1 = persons.iterator();
        while (it1.hasNext()) {
            System.out.println("The force is strong on : " + ((Person)it1.next()).getName());
        }
    }
}
```


Algunas de las operaciones que hemos utilizado tienen más de una sobrecarga. Por lo tanto, sería recomendable que, antes de utilizar alguna, mirarais si alguna de las sobrecargas existentes son lo suficientemente adecuadas para poder utilizarse.

2.4. Las excepciones

Una **excepción** se puede definir como la ocurrencia de un acontecimiento inesperado durante la ejecución normal de un programa (por ejemplo, un error).

Una excepción nos sirve para comunicar la ocurrencia de acontecimientos sin tener que definir un protocolo concreto. Nosotros nos centraremos en el uso de excepciones para el control de errores.

Dado que en Java todo son clases, necesitamos una clase que represente la excepción y un mecanismo del lenguaje que nos permita capturar los errores y tratarlos en caso de necesidad.

Por el hecho de ser objetos, las excepciones nos ofrecen un nuevo abanico de posibilidades para el tratamiento de errores, ya que nos permiten encapsular todos los datos referentes al error dentro de una estructura y trabajar con ellos de manera más sencilla. Si una excepción no es tratada, la ejecución del programa se aborta inmediatamente. Por ello es importante tratarlas todas. 

2.4.1. Creación de una excepción

En Java, cualquier excepción debe heredar de la clase `Exception`, que ya tenemos definida e implementada en la API.

Esta nueva clase debe tener como mínimo la siguiente estructura:

`MyException.java`

```
public class MyException extends Exception {  
}
```

Como podemos observar, no hay que incluir ningún atributo o método, pero entonces tenemos una excepción que no transmite ninguna información adicional a la que nos ofrece la propia clase `Exception`.

Como la propia clase `Exception` ya tiene un atributo de tipo `String` para almacenar el mensaje que dará, una buena costumbre es definir mensajes personalizados que informen de la misma manera sobre el mismo error y, en caso de que fuera necesario, algún otro atributo para poder pasar alguna información que sea útil después para tratar los errores.

Nuestra excepción quedaría de la siguiente manera:

`MyException.java`

```
public class MyException extends Exception {  
    public static String MyMessage = "Write here what you need.";  
    public MyException(String message) {  
        super(message);  
    }  
}
```


Fijaos en que podemos añadir cualquier atributo y método que se nos ocurra, pero normalmente con esta definición ya tendremos suficiente para nuestras aplicaciones.

El método `super` invoca al método constructor de la clase padre.

En caso de que queramos declarar varios métodos constructores, u otros métodos para consultar los atributos que tenga nuestra excepción, simplemente los declararemos de la misma manera que en una clase cualquiera.

2.4.2. Lanzamiento de excepciones

Una vez que tengamos definida una clase que representa nuestras excepciones, habrá que poder utilizarlas para indicar las situaciones anómalas que se producen durante la ejecución del código.

Se indica que se ha producido una excepción con la instrucción siguiente:

```
throw new MyException("Error description");
```

En este caso estamos lanzando una excepción del tipo `MyException` con el texto dado. Para lanzar una excepción hemos de crear una instancia de la clase `Exception` con los parámetros necesarios y, posteriormente, con el uso de la palabra reservada `throw` indicamos que se ha producido el error indicado.

Cabe tener en cuenta que la ejecución del método en el que se ha lanzado la excepción queda interrumpido, e inmediatamente volveremos al lugar donde se realizó el llamamiento, de la misma manera que si hubiéramos realizado un `return` en nuestro código.

2.4.3. Tratamiento de excepciones

Hasta ahora hemos visto cómo podemos crear excepciones y lanzarlas. Ahora debemos aprender cómo podemos tratarlas en caso de que éstas se hayan producido en métodos que nosotros utilizamos en nuestro código.

Para poder tratar las excepciones correctamente hay que tener en cuenta el lugar en el que se pueden producir, es decir, es necesario que tengamos documentados todos los métodos de las clases que utilicemos, y que esta documentación informe de las excepciones que puede generar.

Dado esto, solamente tenemos que incluir dentro de un bloque de código especial el conjunto de instrucciones que pueden generar una excepción, de manera que la ocurrencia de las excepciones se detectará sin que el programa aborte.

Este bloque especial de instrucciones se denomina bloque `try...catch`, y tiene la estructura siguiente:

```
try {  
    // Code that could throw an exception  
}  
catch (ExceptionType exceptionVar) {  
    // Code that handles an exception  
}
```

Escribiremos el código que puede generar una excepción entre las llaves que siguen a la palabra reservada `try`. Dentro de la sentencia `catch` indicaremos qué tipo de excepción se puede generar; le daremos un nombre de variable (para poder recuperar datos más tarde) y escribiremos el código que tratará esta excepción.

Cuando cualquiera de los métodos dentro del bloque `try` lance una excepción del tipo declarado al bloque `catch`, el programa pasará a ejecutar inmediatamente el código dentro de este bloque.

```
private static void createException() throws MyException {  
    throw new MyException(MyException.MyPersonalMessage);  
}  
  
try {  
    createException();  
}  
catch (MyException m) {  
    // Code that handles an exception  
    System.out.println(m.getMessage());  
}
```

Para el tratamiento de excepciones podemos poner cualquier código que necesitemos: puede ser la escritura en un fichero, la pantalla, o simplemente actualizar un contador; todo dependerá de nuestra aplicación y sus necesidades.